CATEGORY THEORY for FUNCTIONAL PROGRAMMING

Mike Spivey

OGI, Winter 1990

Course Plan

(I) Learn basic category theory

* Functors
* Natural transformations
* Categories
* Adjunctions

(II) Explore applications to functional programming

* programming by transformation using laws of a data type
* monads as models of computation

(III) Survey applications in other areas of computing science

* colimit computations (Rydeheard/Burstall)
* initial algebra semantics (ADJ)
* structured specifications (Burstall/Goguen)

1/ Some basic notations

We'll use a dialect of Miranda for expressing functional programs

* polymorphic types
* defining functions by recursion
* lazy evaluation (when needed)
* infix operators and sections
* new types defined by ::=

Reference:

R.S. Bird and P. L. Wadler
Introduction to Functional Programming
Prentice-Hall International 1988

1.1 **Lists** `[1,2,3] : list num`

`[[`b`, `y`, `e`], [`b`, `y`, `e`]] : list (list char)`

`[] : list α` the empty list

`[[]] : list (list α)` a singletun

1.2 **Length** $\#[x_1, x_2, \ldots, x_n] = n$ // def #

$\# : \text{list } \alpha \longrightarrow \text{num}$

Recursive definition:

```
> # [] = 0
> # (x:a) = 1 + # a
```

1.3 **Concatenation**

$[x_1, x_2, \ldots, x_n] +\!\!+ [y_1, y_2, \ldots, y_m]$

$= [x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m]$

$(+\!\!+) : \text{list } \alpha \times \text{list } \alpha \longrightarrow \text{list } \alpha$

* an infix operator

Note • We write (#) to name the operator itself (without applying it to arguments)
• We do not use currying as in real Miranda, but we'll use sections (see below) to get its effect
• a # b means exactly (#) (a, b)

Laws about #:

```
[] # a = a = a # []      //[]#, #[]
(a # b) # c = a # (b # c) //#-assoc
#(a # b) = #a + #b       //#-#
```

Recursive definition:

```
> [] # b = b             // def #
> (x:a) # b = x : (a # b) //def #
```


1.4 We can prove these laws by structural induction on lists: for example, here is the proof of the law #(a # b) = #a + #b:

Proof by induction on a.

Base

```
  #([] # b)
= {def #}
  #b
= {arithmetic}
  0 + #b
= {def #}
  #[] + #b
```

Step if #(a # b) = #a + #b then

```
  #((x:a) # b)
= {def #}
  #(x:(a # b))
= {def #}
  1 + #(a # b)
```

$$
\begin{aligned}
& 1 + \#(a +\!\!+ b) \\
= & \quad \{ \text{induction hypothesis} \} \\
& 1 + (\#a + \#b) \\
= & \quad \{ \text{addition associative} \} \\
& (1 + \#a) + \#b \\
= & \quad \{ \text{def } \# \} \\
& \#(x:a) + \#b. \qquad \square
\end{aligned}
$$

Proof by induction is sound because of the definition of the list type:

(i) [] is a list

(ii) if a is a list and x is an object, then $(x:a)$ is a list

(iii) "nothing else is a list"

We interpret (iii) to mean that any set which contains [] and is closed under (:) contains all lists.

1.5

Map - a higher-order function

$$f * [x_1, x_2, \ldots, x_n] = [f\,x_1, f\,x_2, \ldots, f\,x_n] \quad \text{// def } *$$

$$(*) : (\alpha \to \beta) \times \text{list } \alpha \to \text{list } \beta$$

Recursive definition:

$$> f * [] = []$$

$$> f * (x:a) = (f\,x) : (f * a)$$

---

aside

Sections. If $(\oplus) : X \times Y \to Z$, then we write $(x \oplus)$ for the function which maps any y to $x \oplus y$. In fact,

$$(x \oplus) = \text{curry } (\oplus)\ x$$

where $\text{curry} : (\alpha \times \beta \to \gamma) \to \alpha \to \beta \to \gamma$ is defined by $\text{curry}\ f\ x\ y = f\,(x, y)$.

So
$$
\begin{aligned}
(x \oplus)\ y &= \text{curry } (\oplus)\ x\ y \\
&= (\oplus)(x, y) \\
&= x \oplus y
\end{aligned}
$$

Similarly, $(\oplus\, y)$ is the function which maps any x to $x \oplus y$.
We omit the brackets from $(x \oplus)$ and $(\oplus\, y)$ when the fancy takes us.

Laws about $*$:

$$f * [x] = [f x] \qquad \text{// } * \text{-sing}$$

$$f * (a +\!\!+ b) = (f * a) +\!\!+ (f * b) \qquad \text{// } * \text{-}+\!\!+$$

The following two laws involve the identity function $id : \alpha \to \alpha$ and the composition operator $(\cdot) : (\beta \to \gamma) \times (\alpha \to \beta) \to \alpha \to \gamma$:

$$id * a = a \qquad \text{// id-}*$$

$$(g \cdot f) * a = g * (f * a) \qquad \text{// } \cdot\text{-}*$$

The second of these is true because, taking $a = [x_1, x_2, \ldots, x_n]$,

$$\begin{aligned}
&(g \cdot f) * [x_1, x_2, \ldots, x_n] \\
&= \{\text{def } *\} \\
&[(g \cdot f)\, x_1,\ (g \cdot f)\, x_2,\ \ldots,\ (g \cdot f)\, x_n] \\
&= \{\text{def } \cdot\} \\
&[g(f\, x_1),\ g(f\, x_2),\ \ldots,\ g(f\, x_n)] \\
&= \{\text{def } *\} \\
&g * [f\, x_1,\ f\, x_2,\ \ldots,\ f\, x_n]
\end{aligned}$$

$$= \{\text{def } *\}$$

$$g * (f * [x_1, x_2, \ldots, x_n]).$$

Both laws may be proved more obscurely by structural induction.

Using sections (see "aside") we can write the two laws as

$$(id\ *) = id \qquad \text{// id-}*$$

$$((g \cdot f)\ *) = (g*) \cdot (f\ *) \qquad \text{// } \cdot\text{-}*$$

1.6 Reduce If $(\oplus) : X \times X \to X$ is associative with identity $e : X$, we define $\oplus/ : \text{list}\ X \to X$ so that

$$\begin{aligned}
&\oplus/\ [x_1, x_2, \ldots, x_n] \\
&\quad = x_1 \oplus x_2 \oplus \cdots \oplus x_n \qquad \text{// def-}\oplus/
\end{aligned}$$

$$\oplus/\ [\,] = e. \qquad \text{// } \oplus/[\,]$$

For example, $(+\!\!+)$ is associative with identity $[\,]$, so $+\!\!+/ : \text{list}\ (\text{list}\ \alpha) \to \text{list}\ \alpha$:

$$\begin{aligned}
&+\!\!+/\ [a_1, a_2, \ldots, a_n] \\
&\quad = a_1 +\!\!+ a_2 +\!\!+ \cdots +\!\!+ a_n
\end{aligned}$$

Laws about /:

$$\oplus/\,[x] = x \qquad \text{// } \oplus/\text{-sing}$$

$$\oplus/\,(a +\!\!+ b) = (\oplus/a) \oplus (\oplus/b) \qquad \text{// } \oplus/\text{-}+\!\!+$$

Generalizing this,

$$\begin{aligned}
&\oplus/\,(+\!\!+/\,[a_1, a_2, \ldots, a_n]) \\
&= \{\text{def } +\!\!+/\} \\
&\oplus/\,(a_1 +\!\!+ a_2 +\!\!+ \cdots +\!\!+ a_n) \\
&= \{\oplus/ - +\!\!+\} \\
&(\oplus/a_1) \oplus (\oplus/a_2) \oplus \cdots \oplus (\oplus/a_n) \\
&= \{\text{def } \oplus/\} \\
&\oplus/\,[\oplus/a_1, \oplus/a_2, \ldots, \oplus/a_n] \\
&= \{\text{def } *\} \\
&\oplus/\,((\oplus/) * [a_1, a_2, \ldots, a_n])
\end{aligned}$$

That is, $(\oplus/) \cdot (+\!\!+/) = (\oplus/) \cdot ((\oplus/)*)$

Better: $\oplus/ \cdot +\!\!+/ = \oplus/ \cdot \oplus/*$ // $\oplus/$-prom

Also $f * (+\!\!+/\,[a_1, a_2, \ldots, a_n])$

$$\begin{aligned}
&= \{\text{def } +\!\!+/\} \\
&f * (a_1 +\!\!+ a_2 +\!\!+ \cdots +\!\!+ a_n) \\
&= \{* - +\!\!+\} \\
&(f * a_1) +\!\!+ (f * a_2) +\!\!+ \cdots +\!\!+ (f * a_n) \\
&= \{\text{def } +\!\!+/\} \\
&+\!\!+/\,[f * a_1, f * a_2, \ldots, f * a_n] \\
&= \{\text{def } *\} \\
&+\!\!+/\,((f*) * [a_1, a_2, \ldots, a_n])
\end{aligned}$$

That is, $f* \cdot +\!\!+/ = +\!\!+/ \cdot (f*)*$ // $*$-prom.

Recursive definition: we define $\oplus/$ = fold $(\oplus)$ e where fold has the recursive definition

> fold $(\oplus)$ e [] = e

> fold $(\oplus)$ e (x : a) = x $\oplus$ (fold $(\oplus)$ e a)

i.e. fold : $(\alpha \times \alpha \to \alpha) \to \alpha \to \text{list}\,\alpha \to \alpha$.

1.7

Example The length function # can be given the following alternative definition:

$$\# = +/ \cdot (K\ 1)*$$

where K is the constant function defined by $K\ x\ y = x$.

E.g. $\# [a, b, c, d]$

$= (+/ \cdot (K\ 1)*)\ [a, b, c, d]$

$= +/ ((K\ 1)*\ [a, b, c, d])$

$= +/ [K1\ a, K1\ b, K1\ c, K1\ d]$

$= +/ [1, 1, 1, 1]$

$= 1+1+1+1$

$= 4.$

Consider the problem of finding the total number of elements in a list of lists. One method is to concatenate the lists, then count the elements in the result:

$$count = \# \cdot +/$$

E.g. count $[[b, y, e], [b, y, e]]$

$= \#\ (+/\ [[b, y, e], [b, y, e]])$

$= \#\ [b, y, e, b, y, e]$

$= 6$

A more efficient method might be to count the lists separately, then add the answers:

$$count = +/ \cdot \#*$$

E.g. count $[[b, y, e], [b, y, e]]$

$= +/ (\#*\ [[b, y, e], [b, y, e]])$

$= +/ [\#[b, y, e], \#[b, y, e]]$

$= +/ [3, 3]$

$= 6$

Using our laws about the list operators, we can transform our original program into the more efficient one:

$$
\begin{aligned}
& \#\ \cdot\ \#/ \\
= & \quad \{ \text{def } \# \} \\
& +/\ \cdot\ (K\ 1) *\ \cdot\ \#/ \\
= & \quad \{ * \text{-promote} \} \\
& +/\ \cdot\ \#/\ \cdot\ ((K\ 1) *) * \\
= & \quad \{ +/ \text{-promote} \} \\
& +/\ \cdot\ (+/) *\ \cdot\ ((K\ 1) *) * \\
= & \quad \{ \cdot - * \} \\
& +/\ \cdot\ (+/\ \cdot\ (K\ 1) *) * \\
= & \quad \{ \text{def } \# \} \\
& +/\ \cdot\ \# *
\end{aligned}
$$

□

Note: it isn't necessary to use the definition

$$\# = +/\ \cdot\ (K\ 1) *$$

as the implementation of # - we

can go on using the recursive definition, if that is more efficient, or use a built-in # if one is provided. The definition in terms of +/ and * is useful because it helps with program transformation - compare the argument above with the inductive proof overleaf.

This suggests the following strategy for program development:

- write a clear and simple program using higher-order functions like * and /.
- transform it using laws to improve efficiency or introduce parallelism.
- if necessary, calculate recursive definitions of the functions which result.

As an example, we can calculate a recursive definition of #:

$$
\begin{aligned}
\#\ [\,] &= +/\ ((K\ 1) *\ [\,]) \\
&= +/\ [\,] = 0
\end{aligned}
$$

$$
\begin{aligned}
\#\,(x:a) &= +/\,((K\,1) * (x:a)) \\
&= +/\,((K\,1\;x) : ((K\,1) * a)) \\
&= 1 + (+/\,((K\,1) * a)) \\
&= 1 + \#a\,.
\end{aligned}
$$

For comparison with the transformation above, here is an proof by induction of the identity

$$\#\,(+/\,s) = +/\,(\#* s)$$

Base

$$
\begin{aligned}
&\#\,(+/\,[\,]) \\
=\;& \{\; +/[\,]\;\} \\
&\#\,[\,] \\
=\;& \{\text{def } \#\} \\
&0 \\
=\;& \{\; +/[\,]\;\} \\
&+/\,[\,] \\
=\;& \{\text{def } *\} \\
&+/\,(\#* [\,])
\end{aligned}
$$

Step If $\#\,(+/\,s) = +/\,(\#* s)$, then

$$
\begin{aligned}
&\#\,(+/\,(a:s)) \\
=\;& \{\text{def } +/\;\} \\
&\#\,(a + (+/\,s)) \\
=\;& \{\; \#-+\;\} \\
&\#a + \#\,(+/\,s) \\
=\;& \{\text{ind. hyp.}\} \\
&\#a + (+/\,(\#* s)) \\
=\;& \{\text{def } +/\;\} \\
&+/\,((\#a) : (\#* s)) \\
=\;& \{\text{def } *\} \\
&+/\,(\#* (a:s))
\end{aligned}
$$

□

The direct proof is shorter, contains fewer variables, and is free of the left-to-right bias of the inductive proof.

1.8 Recursive types

The definition

$$\text{btree}\ \alpha ::= \text{Tip}\ \alpha \mid \text{Fork}\ (\text{btree}\ \alpha \times \text{btree}\ \alpha)$$

introduces a type constructor btree, so that for each type X, btree X is a type, such that

(i) if $x : X$ then $\text{Tip}\ x : \text{btree}\ X$

(ii) if $l, r : \text{btree}\ X$ then $\text{Fork}(l, r) : \text{btree}\ X$

(iii) "nothing else is a tree"

We interpret (iii) to mean that if S is a subset of btree X with the properties that $\text{Tip}\ x \in S$ for each $x \in X$ and that $\text{Fork}\ (l, r) \in S$ for each $l, r \in S$, then $S = \text{btree}\ X$.

Equivalent to this induction principle is the "primitive recursion principle" that the equations

$$f\ (\text{Tip}\ x) = u(x)$$

$$f\ (\text{Fork}\ (l, r)) = v(f\ l, f\ r)$$

(where $u : X \to Y$ and $v : Y \times Y \to Y$) uniquely define a function $f : \text{btree}\ X \to Y$.

The induction principle justifies proof by induction: to show that a property $P(t)$ holds of all btrees t, we consider $S = \{ t : \text{btree}\ X \mid P(t) \}$ and show that it has the two properties listed. This amounts to proving

(i) that $P(\text{Tip}\ x)$ holds for all $x : X$

(ii) if $P(l)$ and $P(r)$ hold for any $l, r : \text{btree}\ X$, then $P(\text{Fork}\ (l, r))$ holds also.

Likewise, the recursion principle justifies defining functions by recursion on the structure of trees.

Other examples of recursive type definitions:

$$\text{list}\ \alpha ::= \text{Nil} \mid \text{Cons}\ (\alpha \times \text{list}\ \alpha)$$

or maybe

$$\text{list}\ \alpha ::= [] \mid \alpha : \text{list}\ \alpha.$$

"Recursive" type definitions need not be recursive at all:

$$\alpha \uplus \beta ::= \text{Pink}\ \alpha \mid \text{Blue}\ \beta$$

A value of type $X \uplus Y$ is either Pink x where $x : X$ or Blue y where $y : Y$ — so $X \uplus Y$ is the "disjoint union" of X and Y. The "recursion" principle says that the equations

$$f\ (\text{Pink}\ x) = u\ x$$

$$f\ (\text{Blue}\ y) = v\ y$$

uniquely define a function $f : X \uplus Y \to Z$ in terms of $u : X \to Z$ and $v : Y \to Z$.

Summary

We've defined various functions on lists and seen how they obey laws. These laws are equations and can be used as transformation rules in program development.

We've also looked at recursive type definitions, which let us define various new types — different kinds of trees, disjoint unions and so on.

How can we organize our "theory of lists"? How can we tell if we have enough laws or if the ones we have are redundant? How can we develop a theory of trees, etc.? These are the questions we'll begin to answer now.

References: Books

S. Mac Lane. Categories for the Working Mathematician. Springer Verlag 1971

- the essential text for serious students of category theory

Arbib & Manes. Arrows, Structures and Functors: the categorical imperative. Academic Press

- contains some examples from automata theory (but they don't amount to much!)

J. Lambek & P.J. Scott. Introduction to Higher Order Categorical Logic. Cambridge University Press 1986

- contains useful material on monads.

Papers

R.S. Bird, "An introduction to the theory of lists"

R.S. Bird, "A calculus of functions for program development"

- contain a nicely worked out theory of the list data type.

J.M. Spivey, "A functional theory of exceptions" to appear in Science of Computer Programming

- contains a categorical theory of exceptions.

2/ Type constructors are functors

In this lecture, we begin to explore patterns in the laws about lists - patterns which stress the similarities with other 'type constructors' such as btree, ⊎, × rather than what is special to lists

2.1 list as a functor

If $f : X \longrightarrow Y$
then $f* : \text{list } X \longrightarrow \text{list } Y$.
This suggests a close connection between the type constructor list and the mapping operator * - as list acts on types, so * acts on functions from one type to another.

Also, * obeys two laws that relate it to identity functions and composition:

$$id_X * = id_{\text{list } X}$$

$$(g \cdot f)* = g* \cdot f*$$

We write id_T for the identity function $T \to T$.

Because of these laws, we say that list and * together form a functor.

Definition (version 0)
A functor associates with each type X a type FX and, with each function $f : X \longrightarrow Y$ a function $Ff : FX \longrightarrow FY$ in such a way that

$$F(id_X) = id_{FX}$$

$$F(g \cdot f) = Fg \cdot Ff$$

□

In our example, we had $FX = \text{list } X$ and $Ff = f*$

It's worth checking the types in the second equation. If $f : X \to Y$ and $g : Y \longrightarrow Z$ then

$$g \cdot f : X \to Z,$$

so

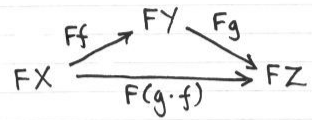
$$F(g \cdot f) : FX \longrightarrow FZ$$

On the other hand, $Ff : FX \longrightarrow FY$ and $Fg : FY \longrightarrow FZ$, so

$$Fg \cdot Ff : FX \longrightarrow FZ$$

also.

We can draw the following diagram of functions and types



The law $F(g \cdot f) = Fg \cdot Ff$ says that it doesn't matter which route we take to get from FX to FZ – both represent the same function. We say that the diagram commutes.

2.2 Another example

Define pair $\alpha == \alpha \times \alpha$
To make this a functor, we need to find for each function $f: X \to Y$ a function – lets call it $f^{(2)}$ – from pair X to pair Y. Lets try defining

$$f^{(2)} : (\alpha \times \beta) \times \text{pair } \alpha \longrightarrow \text{pair } \beta$$

$$f^{(2)}(x, y) = (f\,x,\ f\,y)$$

The types are right: what about the laws? We need

$$id_X^{(2)} = id_{\text{pair } X}$$

$$(g \cdot f)^{(2)} = g^{(2)} \cdot f^{(2)}.$$

The proof is an application of extensionality*:

$$\begin{aligned} & id_X^{(2)}\,(x, y) \\ = & \ (id_X\,x,\ id_X\,y) \\ = & \ (x, y) \\ = & \ id_{\text{pair } X}\,(x, y) \end{aligned}$$

$$\begin{aligned} & (g \cdot f)^{(2)}(x, y) \\ = & \ ((g \cdot f)\,x,\ (g \cdot f)\,y) \\ = & \ (g(f\,x),\ g(f\,y)) \\ = & \ g^{(2)}(f\,x,\ f\,y) \\ = & \ g^{(2)}(f^{(2)}(x, y)) \\ = & \ (g^{(2)} \cdot f^{(2)})(x, y) \end{aligned}$$

So pair and $^{(2)}$ are a functor.

---

* Extensionality means that if $f_1, f_2 : X \to Y$ satisfy

$$f_1\,x = f_2\,x$$

for all $x : X$, then $f_1 = f_2$

2.3 Yet another example

We defined

```
btree α ::= Tip α
          | Fork (pair (btree α))
```

What is the function
$f\circledast : \text{btree } X \longrightarrow \text{btree } Y$
corresponding to $f : X \longrightarrow Y$?

We can define it by recursion:

$f\circledast\ (\text{Tip } x) = \text{Tip } (f\ x)$

$f\circledast\ (\text{Fork } (l, r)) = \text{Fork } (f\circledast l, f\circledast r)$

To prove the laws, we need extensionality and structural induction: we argue by induction on t that $id_X\circledast t = t$ and $(g\cdot f)\circledast t = g\circledast(f\circledast t)$.

Just as we plan to use higher order operations like $*$ and $\circledast$ to eliminate recursion from programs, we plan to use these laws – proved by induction – to eliminate induction from proofs.

Base

$id_X\circledast(\text{Tip } x) = \text{Tip } (id_X\ x) = \text{Tip } x$

$(g\cdot f)\circledast(\text{Tip } x) = \text{Tip } ((g\cdot f)\ x) = \text{Tip } (g\ (f\ x)) = g\circledast(\text{Tip } (f\ x)) = g\circledast(f\circledast(\text{Tip } x))$

Step If $id_X\circledast l = l$, $id_X\circledast r = r$ and $(g\cdot f)\circledast l = g\circledast(f\circledast l)$, $(g\cdot f)\circledast r = g\circledast(f\circledast r)$

Then

$id_X\circledast(\text{Fork } (l, r)) = \text{Fork } (id_X\circledast l, id_X\circledast r) = \{\text{ind. hyp.}\} = \text{Fork } (l, r)$

And

$(g\cdot f)\circledast(\text{Fork } (l, r)) = \text{Fork } ((g\cdot f)\circledast l, (g\cdot f)\circledast r) = \{\text{ind hyp}\} = \text{Fork } (g\circledast(f\circledast l), g\circledast(f\circledast r)) = g\circledast(\text{Fork } (f\circledast l, f\circledast r))$

□

2.4 Identity functor.

Define $\mathrm{Id}\,\alpha == \alpha$.
If $f : X \to Y$, we want $\mathrm{Id}\,f : \mathrm{Id}X \to \mathrm{Id}Y$.
But that's just $\mathrm{Id}\,f : X \to Y$: why not take $\mathrm{Id}\,f = f$?

Immediately,

$$\mathrm{Id}\,id_X = id_X = id_{\mathrm{Id}X}$$

$$\mathrm{Id}\,(g \cdot f) = g \cdot f = \mathrm{Id}\,g \cdot \mathrm{Id}\,f$$

So this is a functor.

2.5 Constant functors

Define $K_Z\,\alpha = Y$
We need $K_Z\,f : K_Z X \to K_Z Y$
i.e. $K_Z\,f : Z \to Z$: we take $K_Z f = id_Z$

Then

$$K_Z\,id_X = id_Z = id_{K_Z X}$$

$$K_Z(g \cdot f) = id_Z = id_Z \cdot id_Z = K_Z g \cdot K_Z f$$

Another functor.

2.6 Composition of functors

If F and G are two functors, define $(GF)X = G(FX)$ and if $f : X \to Y$, define $(GF)f : (GF)X \to (GF)Y$ by $(GF)f = G(Ff)$. That's obvious, but we should check that GF is really a functor:

$$(GF)id_X = G(F\,id_X) = G\,id_{FX} = id_{G(FX)} = id_{(GF)X}$$

$$(GF)(g \cdot f) = G(F(g \cdot f)) = G(Fg \cdot Ff) = G(Fg) \cdot G(Ff) = (GF)g \cdot (GF)f.$$

As you can guess, we're going to write GFX from now on ....

The fact that we can compose functors to get new functors means that we can make complex type constructors act on functions just as simple ones do.

2.7 Functors of 2 (or more) arguments

Earlier we looked at the functor pair which made the type of pairs (x, y) where x and y have the same type. Can we remove this restriction?

If $f : X \longrightarrow U$ and $g : Y \longrightarrow V$, define $f \underline{\text{cross}}\ g : X \times Y \longrightarrow U \times V$ by

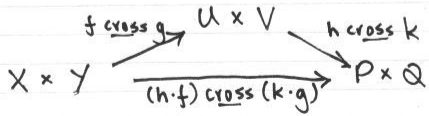
$$(f \underline{\text{cross}}\ g)(x, y) = (f\,x,\ g\,y)$$

We find that

$$id_X \underline{\text{cross}}\ id_Y = id_{X \times Y}$$

and if $f : X \longrightarrow U$; $g : Y \longrightarrow V$; $h : U \longrightarrow P$; $k : V \longrightarrow Q$ then

$$(h \cdot f) \underline{\text{cross}}\ (k \cdot g) = (h \underline{\text{cross}}\ k) \cdot (f \underline{\text{cross}}\ g)$$



Definition (version 0)

A two-argument functor F associates with each pair of types X, Y a type $F(X,Y)$ and with each pair of functions $f : X \longrightarrow U$ and $g : Y \longrightarrow V$ a function $F(f,g) : F(X,Y) \longrightarrow F(U,V)$ so that

$$F(id_X, id_Y) = id_{F(X,Y)}$$

and

$$F(h \cdot f,\ k \cdot g) = F(h,k) \cdot F(f,g). \quad \square$$

Our first example sets $F(X,Y) = X \times Y$ and $F(f,g) = f \underline{\text{cross}}\ g$.

For another example, consider the type $X \uplus Y$ defined by

$$\alpha \uplus \beta ::= \text{Pink}\ \alpha \mid \text{Blue}\ \beta$$

Defining

$$(f \underline{\text{plus}}\ g)(\text{Pink}\ x) = \text{Pink}\ (f\,x)$$

$$(f \underline{\text{plus}}\ g)(\text{Blue}\ y) = \text{Blue}\ (g\,y)$$

gives $f \underline{\text{plus}}\ g : X \uplus Y \longrightarrow U \uplus V$ if $f : X \longrightarrow Y$ and $g : Y \longrightarrow V$.

By case analysis we find that
$(id_X \text{ plus } id_Y)(Pink\ x)$
$= Pink\ (id_X\ x) = Pink\ x$
$= id_{X \oplus Y}\ (Pink\ x)$

(similarly for Blue y)

and $((h \cdot f) \text{ plus } (k \cdot g))\ (Pink\ x)$
$= Pink\ ((h \cdot f)\ x)$
$= Pink\ (h\ (f\ x))$
$= (h \text{ plus } k)\ (Pink\ (f\ x))$
$= (h \text{ plus } k)\ ((f \text{ plus } g)\ (Pink\ x))$.

So this is a functor of two arguments also.

2.8 Recursive types

In some generality, a recursive type definition looks like this:

$$\begin{aligned} T\alpha ::= & \ C_1\ (F_1(\alpha, T\alpha)) \\ | & \ C_2\ (F_2(\alpha, T\alpha)) \\ | & \ \ldots \\ | & \ C_n(F_n(\alpha, T\alpha)) \end{aligned}$$

The C_i's are the constructors and the F_i's are two-argument functors.

[For btree, we take $n = 2$
$C_1 = Tip$ $\quad F_1(X,Y) = X$
$C_2 = Fork$ $\quad F_2(X,Y) = pair\ Y$]

To make T a functor, we need to define $Tf : TX \to TY$ for every $f : X \to Y$. This is achieved by the following recursive definition:

$$\begin{aligned} Tf \cdot C_1 &= C_1 \cdot F_1(f, Tf) \\ Tf \cdot C_2 &= C_2 \cdot F_2(f, Tf) \\ &\ldots \\ Tf \cdot C_n &= C_n \cdot F_n(f, Tf) \end{aligned}$$

or, in Miranda,

$Tf\ (C_1\ w_1) = C_1\ (F_1\ (f, Tf)\ w_1)$
etc.

[For btree, the equations are

$f \circledast (Tip\ x) = Tip\ (f\ x)$ ← $F_1(f,g) = f$

$f \circledast (Fork\ (l,r)) = Fork\ ((f \circledast)^{(2)}\ (l,r))$
← $F_2(f,g) = g^{(2)}$]

General trees with labelled nodes and lists of subtrees:

$$\text{tree}\,\alpha ::= \text{Node}\ (\alpha \times \text{list}(\text{tree}\,\alpha))$$

Here $n = 1$ $\quad C_1 = \text{Node}$ $\quad F_1(X,Y) = X \times \text{list}\,Y$.

If $f : X \to Y$ we define $f\dagger : \text{tree}X \to \text{tree}Y$ by

$$f\dagger(\text{Node}(x, a)) = \text{Node}\ (fx,\ (f\dagger) * a)$$

because $F_1(f,g) = f$ cross $g*$.

2.9 Contravariance

Let's say a predicate on X is a function from X to bool:

$$\text{pred}\,\alpha == \alpha \to \text{bool}.$$

If we try to make this into a functor, something goes wrong.

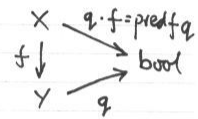
Suppose $f : X \to Y$ and $p : \text{pred}\,X$. The "natural" thing to say is that $(\text{pred}\,f)\,p : \text{pred}\,Y$ holds of some y if and only if some $x : X$ is s.t. $fx = y$ and $px = \text{True}$.

But there's no effective way of finding whether such x exist. Instead, we have to be content with the following:

If $f : X \to Y$ and $q : \text{pred}\,Y$, then $(\text{pred}\,f)\,q : \text{pred}\,X$ is defined by

$$(\text{pred}\,f)\,q\ x = q\,(f\,x)$$

in other symbols,



$$(\text{pred}\,f)\,q = q \cdot f.$$

So $\text{pred}\,f : \text{pred}\,Y \to \text{pred}\,X$ - the arrow goes the wrong way. We say pred is a contravariant functor.

Modified forms of the two laws hold:

$$(\text{pred}\,id_X)\,p = p \cdot id_X = p = id_{\text{pred}\,X}\,p$$

so $\text{pred}\,id_X = id_{\text{pred}\,X}$. Also

$$\begin{aligned}\text{pred}\,(g \cdot f)\,r &= r \cdot g \cdot f \\ &= \text{pred}\,f\ (\ \cdot \\ &= \text{pred}\,f\ (\text{pred}\,g\ r)\end{aligned}$$

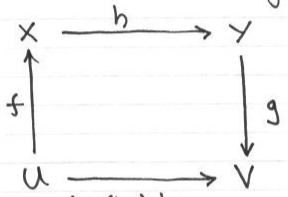
so $\text{pred}\,(g \cdot f) = \text{pred}\,f \cdot \text{pred}\,g$

← BACKWARDS.

Actually, $\rightarrow$ itself is a functor of two arguments, contravariant in the first argument and covariant in the second. To see this, define

$$fun(\alpha, \beta) == \alpha \rightarrow \beta$$

If $f : U \rightarrow X$ and $g : Y \rightarrow V$, we want to define $fun(f,g) : fun(X,Y) \rightarrow fun(U,V)$.



If $h : fun(X,Y)$, we define $fun(f,g)\ h : fun(U,V)$ by

$$fun(f,g)\ h = g \cdot h \cdot f$$

and the two laws hold:

$$fun(id_X, id_Y) = id_{fun(X,Y)}$$

$$fun(g \cdot f, k \cdot h) = fun(f,k) \cdot fun(g,h)$$

3/ Polymorphic functions are natural transformations

A polymorphic function has a type such as

$$\theta_\alpha : F\alpha \longrightarrow G\alpha$$

where F and G are two type constructors. That is, a polymorphic function θ is really a family of functions θ_X, one for each type X.

Miranda and other functional programming languages allow us to omit the subscript X; the type-checking algorithm in the compiler will fill it in for us. But we will write it in explicitly when it seems to help.

The theme of this lecture is that if θ is a "genuine" polymorphic function, then it will obey a law that links it with the type constructors F and G - which we expect will be functors. In categorical language, polymorphic functions like θ are natural transformations.

3.1 concats as a natural transformation

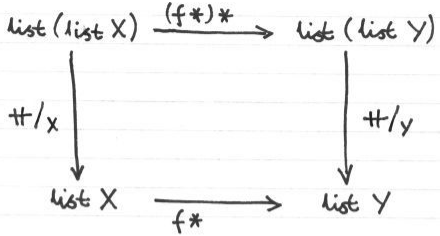
The function $++/\alpha : \text{list}(\text{list}\,\alpha) \longrightarrow \text{list}\,\alpha$ is polymorphic: whatever the type X, $++/_X$ will take lists of lists of X and concatenate all their elements into a single list of X.

If $f : X \longrightarrow Y$ is any function, we can form the functions

$$f* : \text{list}\,X \longrightarrow \text{list}\,Y$$
$$(f*)* : \text{list}(\text{list}\,X) \longrightarrow \text{list}\,Y$$

The first of these applies f to each element of a list of X and makes a list from the results. The second, $(f*)*$ does this to each element of a list of lists: the result is a list of lists with exactly the same shape as the argument, but with each X replaced by the Y to which f maps it.

These two functions form the top and bottom of the following square diagram:



The sides are two instances of the polymorphic function $++/$. We say the square commutes if the composition of the top and right-hand side is equal as a function to the composition of the bottom and the left-hand side, i.e. if

$$++/_Y \cdot (f*)* \;=\; f* \cdot ++/_X$$

This law is true: it is the one we called $*$-promotion.

Because the square commutes for every f, we say $++/$ is a natural transformation and write

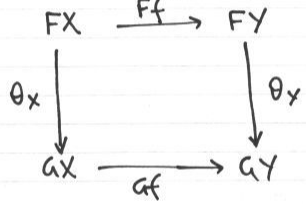
$$++/ : \text{list}\,\text{list} \dot{\longrightarrow} \text{list}$$

3.2 **Definition** (version 0)

A natural transformation $\theta : F \dot{\longrightarrow} G$ from a functor F to a functor G is a family of functions

$$\theta_X : FX \longrightarrow GX,$$

one for each type X, such that the following diagram commutes



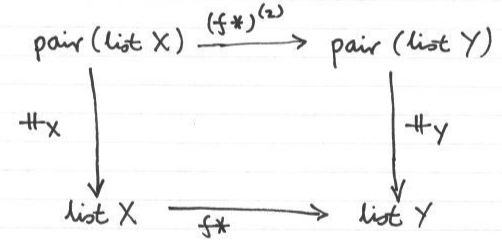
i.e. $\theta_Y \cdot Ff = Gf \cdot \theta_X$, for every function $f : X \longrightarrow Y$.

3.3 **++ as a natural transformation**

The concatenation operator $++_X$ takes two arguments. Or, to put it another way, it takes a pair of lists as its argument:

$$(++_\alpha) : \text{pair}(\text{list}\ \alpha) \longrightarrow \text{list}\ \alpha$$

For ++ to be a natural transformation, here is the square which must commute:

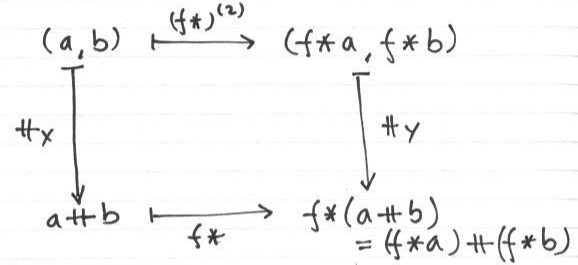


We obtain the function $(f*)^{(2)}$ by applying first the functor list (or $*$) then the functor pair (or (2)) to f.

The square expresses the equation

$$++_Y \cdot (f*)^{(2)} = f* \cdot ++_X$$

To make more concrete sense of this equation, we can take a pair of lists (a, b) and "chase it round the diagram"



So the square commutes if for all $a, b : \text{list } X$,

$$f*(a +\!\!+ b) = (f*a) +\!\!+ (f*b).$$

This is the law $* - +\!\!+$.

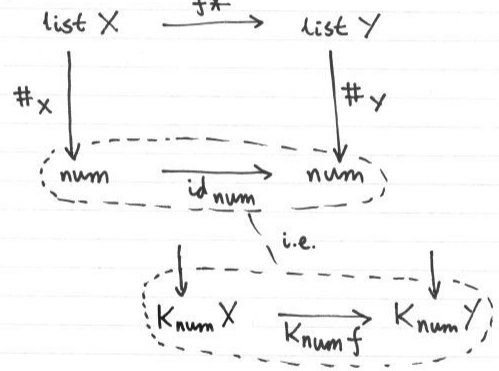
3.4 Other examples

To see other polymorphic functions as natural transformations, we sometimes need to use "trivial" functors like K_Z and Id.

The function $\#_\alpha : \text{list } \alpha \longrightarrow \text{num}$ can be viewed as

$$\#_\alpha : \text{list } \alpha \longrightarrow K_{\text{num}}\, \alpha .$$

If $f : X \longrightarrow Y$, we can draw the square



This commutes if

$$\#_Y \cdot f* \;=\; id_{\text{num}} \cdot \#_X$$

that is, if for all lists a,

$$\#(f* a) = \# a$$

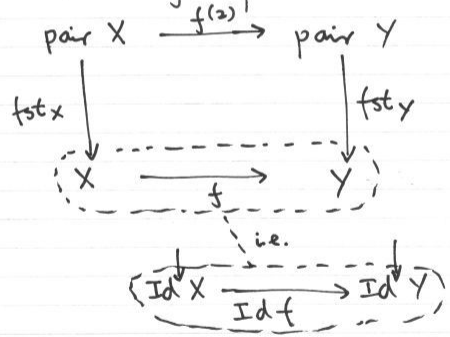
So # is a natural transformation

$$\#: \text{list} \xrightarrow{\cdot} K_{\text{num}}.$$

Again, the function $\text{fst}_\alpha : \text{pair}\,\alpha \to \alpha$ can be viewed as

$$\text{fst}_\alpha : \text{pair}\,\alpha \longrightarrow \text{Id}\,\alpha$$

It is a natural transformation because the following square commutes:

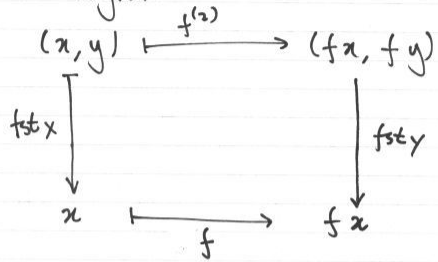


That is, because

$$\text{fst}_Y \cdot f^{(2)} = f \cdot \text{fst}_X$$

for all functions $f : X \to Y$

Chasing a pair (x, y) round the diagram:



we find that this is true, so fst is a natural transformation

$$\text{fst}: \text{pair} \xrightarrow{\cdot} \text{Id}.$$

3.5 Multiple parameters.

Actually, we can do a little better than this with fst, by looking on it as a function

$$\text{fst}_{\alpha\beta} : \alpha \times \beta \longrightarrow \alpha$$

or, writing α explicitly as a functor of α and β,

$$\text{fst}_{\alpha\beta} : \alpha \times \beta \longrightarrow \text{Fst}(\alpha, \beta)$$

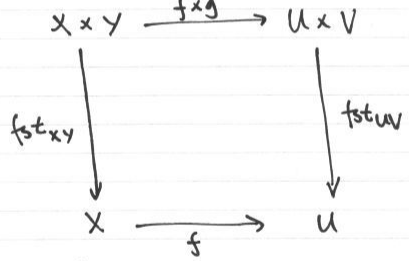
Here Fst is the functor defined by

$$Fst(X,Y) = X$$

and if $f: X \to U$, $g: Y \to V$ then

$$Fst(f,g) = f : Fst(X,U) \to Fst(Y,V).$$

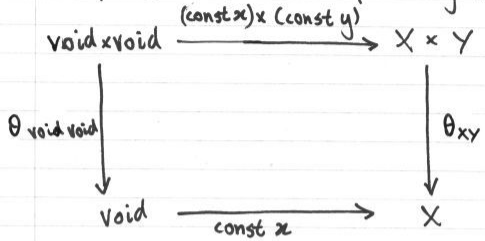
The function fst is a natural transformation $fst: -\times- \dot{\to} Fst$ if the following square commutes:



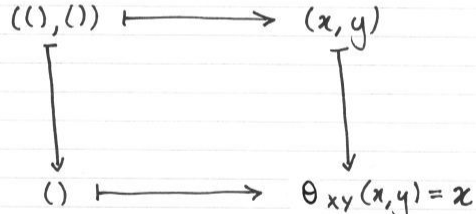
i.e. if $fst_{UV} \cdot (f \times g) = f \cdot fst_{XY}$, a fact easily proved by extensionality.

In fact, the converse is true: fst is the only natural transformation from $-\times-$ to Fst.

For if $\theta : -\times- \dot{\to} Fst$ then chasing $((),())$ round the following diagram:



gives



So instead of the usual definition of fst, we could write the complete specification

$$fst : -\times- \dot{\to} Fst.$$

in purely categorical terms.

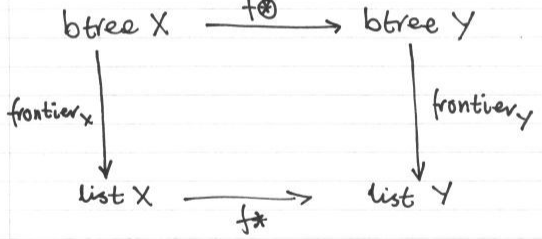
3.6 Binary trees

Define $\text{frontier}_\alpha : \text{btree}\,\alpha \longrightarrow \text{list}\,\alpha$ by

$$\text{frontier}\ (\text{Tip}\ x) = [x]$$

$$\text{frontier}\ (\text{Fork}(t,u)) = \text{frontier}\ t \mathbin{+\!\!+} \text{frontier}\ u.$$

This polymorphic function from trees to lists is a natural transformation:



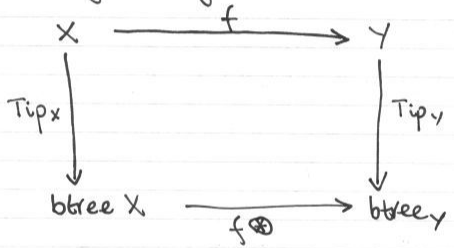
$$\text{frontier}_Y \cdot f\circledast = f* \cdot \text{frontier}_X.$$

Here, $f\circledast$ is the operation which copies a tree, applying f to each leaf. The law says that re-labelling in this way then taking the frontier gives the same result as taking the frontier first, then applying f

to each of its elements.

$$\text{frontier} : \text{btree} \dot{\longrightarrow} \text{list}.$$

Not only are functions defined polymorphically on lists natural transformations; so are the constructors Tip and Fork. For Tip we get the following diagram:



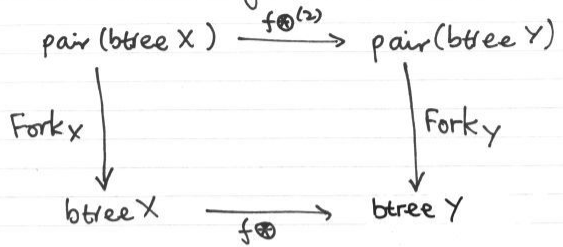
which commutes because

$$\text{Tip}_Y \cdot f = (f\circledast) \cdot \text{Tip}_X$$

or, in other symbols,

$$f\circledast\,(\text{Tip}\ x) = \text{Tip}\ (f\ x).$$

For Fork, we get:



which commutes because

$$\text{Fork}_Y \cdot f^{\circledast(2)} = f^{\circledast} \cdot \text{Fork}_X$$

or – if we chase an arbitrary pair of trees (l, r) round the diagram –

$$f^{\circledast}(\text{Fork}(l,r)) = \text{Fork}(f^{\circledast} l, f^{\circledast} r).$$

The naturality of Tip and Fork is expressed in the same two equations we used to define $f^{\circledast}$ by recursion. So again we have reduced a programming concept to purely categorical terms.

3.7 Composing natural transformations

So far, we've looked at polymorphic functions in isolation. But we can also get new polymorphic functions – and new natural transformations – by putting together old ones in various ways.

"Vertical" composition takes two natural transformations

$$\theta : F \xrightarrow{\cdot} G$$
$$\varphi : G \xrightarrow{\cdot} H$$

and makes a new one

$$\varphi \cdot \theta : F \xrightarrow{\cdot} H.$$

We can take a natural transformation

$$\theta : F \xrightarrow{\cdot} G$$

and a functor H, and make new transformations

$$H\theta : HF \xrightarrow{\cdot} HG$$
$$\theta H : FH \xrightarrow{\cdot} GH$$

– this is "composing a natural transformation with a functor".

Finally, we'll define "horizontal" composition of natural transformations.

If $\theta : F \xrightarrow{\cdot} G$
$\varphi : H \xrightarrow{\cdot} K$
then
$\varphi \circ \theta : HF \xrightarrow{\cdot} KG$

Notice that two n.t.'s must "fit together" for their vertical composition to be defined, but that any two n.t's can be composed horizontally.

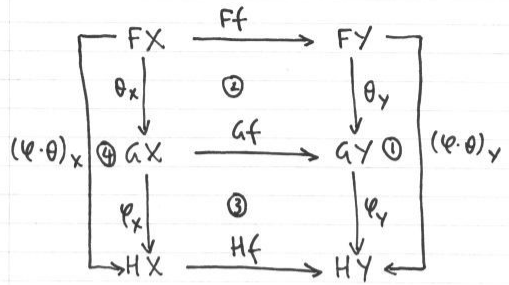
3.8 Vertical composition

If $\theta : F \xrightarrow{\cdot} G$; $\varphi : G \xrightarrow{\cdot} H$, we define $\varphi \cdot \theta : F \xrightarrow{\cdot} H$ by

$$(\varphi \cdot \theta)_X = \varphi_X \cdot \theta_X$$

This is natural because

$$\begin{aligned} & (\varphi \cdot \theta)_Y \cdot Ff \\ = & \{ \text{def } \cdot \} \ ① \\ & \varphi_Y \cdot \theta_Y \cdot Ff \\ = & \{ \theta \text{ natural} \} \ ② \\ & \varphi_Y \cdot Gf \cdot \theta_X \end{aligned}$$



$$\begin{aligned} = & \{ \varphi \text{ natural} \} \ ③ \\ & Hf \cdot \varphi_X \cdot \theta_X \\ = & \{ \text{def } \cdot \} \ ④ \\ & Hf \cdot (\varphi \cdot \theta)_X \end{aligned}$$

An example of vertical composition is composing $frontier_\alpha : btree\ \alpha \longrightarrow list\ \alpha$ with $reverse : list\ \alpha \longrightarrow list\ \alpha$ to get the function

$$reverse \cdot frontier : btree\ \alpha \longrightarrow list\ \alpha$$

which lists the leaves in right-to-left order. The Miranda compiler successfully hides from us the

fact that the polymorphism of frontier and reverse propagates to their composition.

3.9 Composition with functors

If $\theta : F \longrightarrow G$, and H is a functor, we define $H\theta : HF \longrightarrow HG$ and $\theta H : FH \longrightarrow GH$ by

$$(H\theta)_X = H(\theta_X)$$

$$(\theta H)_X = \theta_{HX}$$

To see that $H\theta$ is natural, we take the diagram

$$\begin{array}{ccc} FX & \xrightarrow{Ff} & FY \\ {\scriptstyle \theta_X}\downarrow & & \downarrow{\scriptstyle \theta_Y} \\ GX & \xrightarrow[Gf]{} & GY \end{array}$$

and "apply" H to it:

$$\begin{array}{ccc} HFX & \xrightarrow{HFf} & HFY \\ {\scriptstyle H\theta_X}\downarrow & & \downarrow{\scriptstyle H\theta_Y} \\ HGX & \xrightarrow[HGf]{} & HGY \end{array}$$

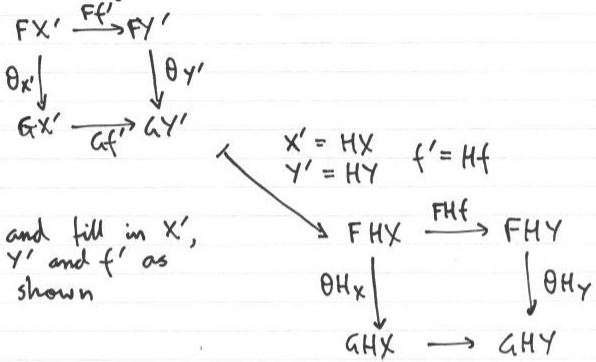
In other words,

$$\begin{aligned} & (H\theta)_Y \cdot HFf \\ = & \quad \{ \text{def } H\theta, \text{def } HF \} \\ & H(\theta_Y) \cdot H(Ff) \\ = & \quad \{ H \text{ is a functor} \} \\ & H(\theta_Y \cdot Ff) \\ = & \quad \{ \theta \text{ is natural} \} \\ & H(Gf \cdot \theta_X) \\ = & \\ & H(Gf) \cdot H(\theta_X) \\ = & \\ & HGf \cdot (H\theta)_X \end{aligned}$$

As an example, we can take the polymorphic function frontier : btree → list and combine it with the functor pair$^{(2)}$ to get

"pair frontier"$_\alpha$: pair(btree α) → pair(list α)

To see that θH is natural, we take the diagram



and fill in X', Y' and f' as shown

This kind of composition is also well-hidden in Miranda, but we can define

```
frontier-pair : btree (pair α)
                → list (pair α)
```

by the equation

```
frontier-pair_α = frontier pair α
```

which we write (in Miranda)

```
> frontier-pair = frontier.
```


3.10

Horizontal composition

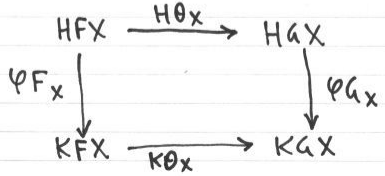
If we have a binary tree with leaves labelled by lists of numbers, we can flip the whole tree with

```
flip : btree → btree
```

or reverse each list with reverse $\circledast$. If we do both, it plainly doesn't matter which order we do them in:

```
flip · reverse ⊛ = reverse ⊛ · flip.
```

Here is the general picture: we have n.t.'s $\theta : F \dot{\to} G$ and $\varphi : H \dot{\to} K$. We can form $H\theta : HF \dot{\to} HG$ and $\varphi G : HG \dot{\to} KG$ and so get the (vertical) composition $\varphi G \cdot H\theta : HF \dot{\to} KG$. Or we can form $K\theta : KF \dot{\to} KG$ and $\varphi F : HF \dot{\to} KF$ and get the composition $K\theta \cdot \varphi F : HF \dot{\to} KG$. The two are the same, because for each X



commutes, since φ is natural. So we can define

$$\varphi \circ \theta = \varphi G \cdot H\theta = K\theta \cdot \varphi F : HF \xrightarrow{\bullet} KG.$$

3.11 Laws of composition

Both $\bullet$ and $\circ$ are associative, and they have identity elements.
If $\theta : F \xrightarrow{\bullet} G$ then

$$Id_G \bullet \theta = \theta = \theta \cdot Id_F$$

and

$$id \circ \theta = \theta = \theta \circ id$$

where $id : Id \xrightarrow{\bullet} Id$ s.t. id_X is the identity arrow $X \longrightarrow X$
and $Id_F : F \xrightarrow{\bullet} F$ is s.t. $(Id_F)_X = id_{FX}$

These operators obey many other laws: see the exercises.

Exercises

(1) Define a function $flip_\alpha : btree\ \alpha \to btree\ \alpha$ which takes the "mirror image" of a binary tree.

(i) What property of flip is expressed by saying it is a natural transformation?

(ii) Prove by structural induction that

reverse · frontier = frontier · flip.

(2) The type of pairs of elements of X can (also) be represented by couple X where

couple α == bool $\to \alpha$.

(i) Define an action of couple on functions which makes it a functor.

(ii) Define first, second : couple $\alpha \to \alpha$ and show that they are natural transformations.

(3) Define a reduction operator λ on binary trees so that if

$$(\oplus) : X \times X \longrightarrow X$$

then $\oplus\lambda$: btree $X \longrightarrow X$

Without further recursion, define

pathtree : btree $\alpha \longrightarrow$ btree (list bool)

the function which replaces each leaf by its path from the root:

(4) If $\theta : F \dot{\longrightarrow} G$ is a natural transformation, define an operation $\hat{\theta}$ which associates with each function $f: X \longrightarrow Y$ a function

$$\hat{\theta}f : FX \longrightarrow GY$$

in such a way that

$$Gg \cdot \hat{\theta}f = \hat{\theta}(g \cdot f) = \hat{\theta}g \cdot Ff \qquad (1)$$

whenever $f: X \longrightarrow Y$; $g: Y \longrightarrow Z$.

Conversely, show that if h is an operation which has the property (1) of $\hat{\theta}$, then $h = \hat{\theta}$ for an unique n.t. θ given by $\theta x = h(id x)$.

(5) Suppose abort$_{\alpha\beta} : \alpha \longrightarrow \beta$ is defined in Miranda by

```
> abort x = abort x
```

Is abort a natural transformation?

(6) (i) If $\theta : F \dot{\longrightarrow} G$; $\varphi : G \dot{\longrightarrow} H$; $\theta' : F' \dot{\longrightarrow} G'$; $\varphi' : G' \dot{\longrightarrow} H'$, prove that

$$(\varphi' \cdot \theta') \circ (\varphi \cdot \theta) = (\varphi' \circ \varphi) \cdot (\theta' \circ \theta)$$

(ii) Show that $\circ$ is associative

(iii) Show that $H\theta = Id_H \circ \theta$, $\theta H = \theta \circ Id_H$ and derive $H(\varphi \circ \theta) = (H\varphi) \circ \theta$ and $(\varphi \circ \theta)H = \varphi \circ (\theta H)$.

4/ Developing Functional Programs by Transformation

R.S. Bird, "Algebraic Identities for Program Calculation", Computer Journal 1989.

5/ Specifications are categories Abstract data types are adjunctions

We now turn our attention to laws like the following:

$$\oplus/ \cdot \square_X = id_X$$

$$+\!\!+/ \cdot \square_X * = id_{list\ X}$$

where $\square_X : X \longrightarrow list\ X$ is defined by $\square_X\ x = [x]$.

More importantly, we'll look at the proof and application of the following "homomorphism theorem" for the list data type:

Homomorphism theorem

If $h : \langle list\ X, +\!\!+, [\,] \rangle \longrightarrow \langle Y, \oplus, e \rangle$ is a homomorphism of monoids [i.e. a function $h : list\ X \longrightarrow Y$ such that $h(a +\!\!+ b) = (h\ a) \oplus (h\ b)$ and $h[\,] = e$] then h can be written in the form

$$h = \oplus/ \cdot f*$$

for a unique $f : X \longrightarrow Y$

5.1 ### Example

An example of a homomorphism is the function

$$lines : list\ CHAR \longrightarrow list^+(list\ CHAR)$$

which maps a text to the (non-empty) list of NL-separated lines in the text.

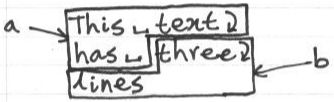
Given a text $a +\!\!+ b$, we can compute its sequence of lines by finding the lines in a and b separately, then combining them

by joining the last line of a with the first line of b:

$$\text{lines}\ (a +\!\!+ b) = (\text{lines}\ a) \otimes (\text{lines}\ b)$$

where $(s +\!\!+ [l]) \otimes ([r] +\!\!+ t) = s +\!\!+ [l +\!\!+ r] +\!\!+ t$

A picture:



lines a = ["This␣text", "has␣"]

lines b = ["three", "lines"]

lines (a++b) = ["This␣text", "has␣three", "lines"]

As it happens, the operation $\otimes$ is associative, and [[]] is an identity element. Of course, lines [] = [[]], so this makes lines a homomorphism

$$\text{lines} : \langle \text{list CHAR}, +\!\!+, [] \rangle \longrightarrow \langle \text{list}^+(\text{list CHAR}), \otimes, [[]] \rangle$$

The homomorphism theorem tells us that lines can be written as

$$\text{lines} = \otimes/ \cdot f*$$

where f: CHAR → list⁺(list CHAR) is given by $f = \text{lines} \cdot [\,]$, i.e.

$$f\,x = \text{lines}\ [x]$$

If $x \neq$ NL then

$$f\,x = \text{lines}\ [x] = [[x]]$$

Also, f NL = lines [NL] = [[], []]

A text containing just NL is considered to contain 2 empty lines

5.2 Categories

We've been making free use of assertions like "M is a monoid" and "h is a homomorphism". Let's focus on the properties of monoids and homomorphisms that are needed to make the homomorphism theorem true (& which we can transfer

to other data types)

Definition A category $\mathcal{C}$ consists of a collection of objects $X \in \mathcal{C}$, and for each pair of objects $X, Y \in \mathcal{C}$ a collection of arrows $\mathrm{Hom}(X, Y)$ from X to Y.

We write $f : X \to Y$ for $f \in \mathrm{Hom}(X, Y)$ and require:

We use curly letters $\mathcal{C}$, $\mathcal{D}$, ... for arbitrary categories, bold-face **Type** for particular examples

(i) if $f : X \to Y$ and $g : Y \to Z$ then there is a composite arrow $g \cdot f : X \to Z$.

(ii) for each object $X \in \mathcal{C}$, there is an identity arrow $id_X : X \to X$

(iii) Composition is associative:

$$h \cdot (g \cdot f) = (h \cdot g) \cdot f$$

whenever defined

(iv) Identity arrows are identity elements for composition

□

Our favourite example is the category **Type**, whose objects are types and whose arrows $f : X \to Y$ are just functions from X to Y. Other examples:

(a) **Monoid** - in which the objects are monoids $M = \langle X, \oplus, e \rangle$ with X a type and $\oplus : X \times X \to X$, $e : X$ with $\oplus$ associative and e an identity element. An arrow $h : M \to N$ is a homomorphism of monoids.

This is a category because
(i) composing homomorphisms gives a new homomorphism

(ii) the identity function id_X is a homomorphism from $M = \langle X, \oplus, e \rangle$ to itself

(iii) "composition of homomorphisms is really just" composition of functions, so it is associative.

(iv) For the same reason, the identity arrows are identities.

(b) If G is a graph, Path(G) is a category with nodes of G as objects and paths in G as arrows - a path being a non-empty sequence of pairwise-adjacent nodes.

Composition of paths is defined when one path finishes where the other starts - the result is the path that follows first one then the other.

The identity arrow for a node x is the singleton path $\langle x \rangle$ that starts at x and stays there.

(c) Trivial categories

$\underline{0}$ - no objects, no arrows

$\underline{1}$ - one object & its identity arrow

$\underline{2}$ - two objects, three arrows

Most of the categories we'll meet will be like (a) in that the objects are types with extra structure and the arrows are functions that preserve that structure. But we'll meet others too.

5.3 Functors and Natural Transformations revisited

The definitions of functor and natural transformation can be generalized to allow functors from one category to another - so far we've looked only at functors from Type to Type.

Definition (Version 1)

A functor $F : \mathcal{C} \longrightarrow \mathcal{D}$ from a category $\mathcal{C}$ to a category $\mathcal{D}$

- maps objects $X \in \mathcal{C}$ to objects $FX \in \mathcal{D}$
- maps arrows $f : X \to Y$ in $\mathcal{C}$ to arrows $Ff : FX \longrightarrow FY$ in $\mathcal{D}$

such that

(i) if $f: X \longrightarrow Y$; $g: Y \longrightarrow Z$ in $\mathcal{C}$, then

$$F(g \cdot f) = Fg \cdot Ff$$

($\cdot$ on the left: composition in $\mathcal{C}$; $\cdot$ on the right: composition in $\mathcal{D}$.)

(ii) $F(id_X) = id_{FX}$

(id_X: identity arrow in $\mathcal{C}$; id_{FX}: identity arrow in $\mathcal{D}$.) □

Definition (version 1)

A natural transformation $\theta: F \dot{\longrightarrow} G$, for $F, G: \mathcal{C} \longrightarrow \mathcal{D}$ functors, associates with each object $X \in \mathcal{C}$ an arrow

$$\theta_X : FX \longrightarrow GX \text{ in } \mathcal{D}$$

such that for each arrow $f: X \longrightarrow Y$ in $\mathcal{C}$, the square

$$\begin{array}{ccc} FX & \xrightarrow{Ff} & FY \\ {\scriptstyle \theta_X}\downarrow & & \downarrow{\scriptstyle \theta_Y} \\ GX & \xrightarrow[Gf]{} & GY \end{array}$$

commutes in $\mathcal{D}$. □

5.4 Examples of functors.

A trivial example of a functor from Monoid to Type is U, defined by

$$U\langle X, \oplus, e\rangle = X$$

and if $h: \langle X, \oplus, e\rangle \longrightarrow \langle Y, \otimes, d\rangle$ in Monoid then

$$Uh = h: X \longrightarrow Y \text{ in Type}$$

The fact that this is a functor reflects the fact that monoids are types with extra structure.

A less trivial example is the functor F: Type $\longrightarrow$ Monoid defined by

$$FX = \langle \text{list } X, +\!\!+, e\rangle$$

and if $f: X \longrightarrow Y$ in Type then

$$Ff = f*: FX \longrightarrow FY \text{ in Monoid}$$

The fact that F is a functor

is expressed by the laws

$$f*(a +\!\!+ b) = (f*a) +\!\!+ (f*b)$$
$$f*[\,] = [\,]$$

- which in fact say that $+\!\!+$ and $[\,]$ are natural transformations. More about this later, maybe.

5.5 Reduction

If $M = \langle X, \oplus, e\rangle$ is a monoid, we can form the monoid

$$FUM = FX = \langle \text{list}X, +\!\!+, [\,]\rangle,$$

and the reduction $\oplus/$ is then a monoid homomorphism

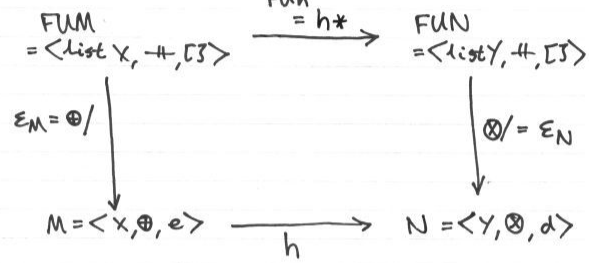
$$\oplus/ : FUM \longrightarrow M.$$

This is one component of a natural transformation $\varepsilon : FU \xrightarrow{\bullet} Id_{\text{Monoid}}$ defined by

$$\varepsilon_M = \oplus/ \quad \text{where } M = \langle X, \oplus, e\rangle$$

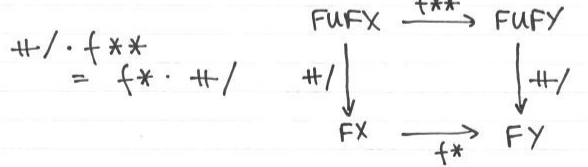
That ε is natural is expressed

in the following square, which can be drawn for any homomorphism $h : M \longrightarrow N$ where $M = \langle X, \oplus, e\rangle$ and $N = \langle Y, \otimes, d\rangle$:



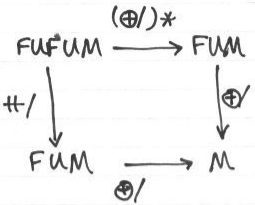
$$\otimes/ \cdot\, h* = h \cdot \oplus/$$

Two special cases are obtained by taking $h = f* : FX \longrightarrow FY$:



and by taking $h = \oplus/ : FUM \longrightarrow M$:

$$\oplus/ \cdot (\oplus/)* = \oplus/ \cdot \#/$$



The diagonal of this last square is $(\varepsilon \circ \varepsilon)_M$.

5.6 Triangle laws

The natural transformation $\varepsilon : FU \dot{\longrightarrow} Id_{Monoid}$ is "almost" an inverse to the natural transformation $\eta : Id_{Type} \dot{\longrightarrow} FU$ given by $\eta_X = \square_X$:

$$\oplus/ \cdot \square_X = id_X$$

i.e. $U(\varepsilon_M) \cdot \eta_{UM} = id_{UM}$

i.e. $\boxed{U\varepsilon \cdot \eta U = Id_U}$

$$\#/ \cdot \square_X * = id_{FX}$$

i.e. $\varepsilon_{FX} \cdot F(\eta_X) = id_{FX}$

i.e. $\boxed{\varepsilon F \cdot F\eta = Id_F}$

Because of these laws, we say that there is an adjunction between Type and Monoid

5.7 Adjunctions

Definition If $\mathcal{X}$ and $\mathcal{A}$ are categories, an adjunction

$$\langle F, U, \eta, \varepsilon \rangle : \mathcal{X} \longrightarrow \mathcal{A}$$

consists of:

The letters $\mathcal{X}, \mathcal{A}$, F, U, η, ε are conventional

(i) Two functors

$F : \mathcal{X} \longrightarrow \mathcal{A}$
$U : \mathcal{A} \longrightarrow \mathcal{X}$

(ii) Two natural transformations

$\eta : Id_{\mathcal{X}} \dot{\longrightarrow} UF$
$\varepsilon : FU \dot{\longrightarrow} Id_{\mathcal{A}}$

such that

$$\boxed{\begin{aligned} U\varepsilon \cdot \eta U &= Id_U \\ \varepsilon F \cdot F\eta &= Id_F \end{aligned}}$$

Memorize it!

□

Exercises (should appear a little later)

(1) Referential transparency

Assume types var and op are given. Define

$$\text{expr}\ \alpha ::= \text{Var}\ \alpha \mid \text{Apply}\ (\text{op} \times \text{list}(\text{expr}\ \alpha))$$

* show how to view this type as an adjunction and state the corresponding homomorphism theorem.

* in terms of apply: op × list num → num (for simplicity, we ignore problems with wrong number of arguments), define

$$\text{eval} : (\text{var} \to \text{num}) \to (\text{expr var} \to \text{num})$$

so that eval ρ E is the result of evaluating E in environment ρ.

* define the substitution operator

$$\text{subst} : (\text{var} \to \text{expr var}) \to (\text{expr var} \to \text{expr var})$$

so that subst σ E is the expression obtained by substituting expressions for variables according to σ.

* prove

$$\text{eval}\ \rho \cdot \text{subst}\ \sigma = \text{eval}\ (\text{eval}\ \rho \cdot \sigma)$$

without explicit induction.

(2) Composition of adjunctions

If $\langle F, U, \eta, \varepsilon \rangle : \mathcal{X} \to \mathcal{A}$
$\langle F', U', \eta', \varepsilon' \rangle : \mathcal{A} \to \mathcal{D}$,
show that there is an adjunction
$\langle F^*, U^*, \eta^*, \varepsilon^* \rangle : \mathcal{X} \to \mathcal{D}$
given by

$$F^* = F'F,$$
$$U^* = UU',$$
$$\eta^* = U\eta'F \cdot \eta,$$
$$\varepsilon^* = \varepsilon' \cdot U'\varepsilon F'.$$

Prove it without ever writing down an object of the categories $\mathcal{X}, \mathcal{A}, \mathcal{D}$!

(3) Product and sum as adjoints

If $\langle F, U, \eta, \varepsilon \rangle : \mathcal{H} \longrightarrow \mathcal{A}$, we say F is left adjoint to U and U is right adjoint to F.

Let Type^2 be the category with objects $\langle X, Y \rangle$ where $X, Y \in \mathrm{Type}$ and arrows

$$\langle f, g \rangle : \langle X, Y \rangle \longrightarrow \langle U, V \rangle$$

where $f : X \longrightarrow U$ and $g : Y \longrightarrow V$. Composition is defined pairwise and the identity arrows are $id_{\langle X,Y \rangle} = \langle id_X, id_Y \rangle$.

Define $\Delta : \mathrm{Type} \longrightarrow \mathrm{Type}^2$ by

$$\Delta X = \langle X, X \rangle$$

and if $f : X \longrightarrow Y$ then

$$\Delta f = \langle f, f \rangle : \langle X, X \rangle \longrightarrow \langle Y, Y \rangle$$

(i) Show that there is a functor $prod : \mathrm{Type}^2 \longrightarrow \mathrm{Type}$ with $prod\langle X, Y \rangle = X \times Y$. that is right adjoint to Δ. What are the unit and counit?

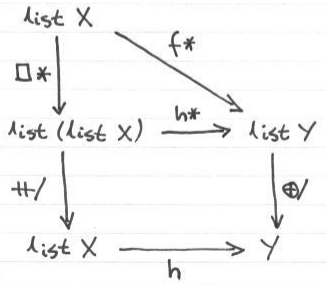
(ii) Similarly, show that there is a functor sum left adjoint to Δ with $sum\langle X, Y \rangle = X \uplus Y$.

5.8

Proof of homomorphism theorem

Suppose $h : FX \longrightarrow M$ is a homomorphism, $M = \langle Y, \oplus, e \rangle$, and set $f = h \cdot \Box$. Then

$$\begin{aligned}
& \oplus/ \cdot f* \\
= & \quad \{ \text{def } f \} \\
& \oplus/ \cdot (h \cdot \Box)* \\
= & \quad \{ * \text{ a functor} \} \\
& \oplus/ \cdot h* \cdot \Box* \\
= & \quad \{ h \text{ a hom, } / \text{ natural} \} \\
& h \cdot +\!\!+/ \cdot \Box* \\
= & \quad \{ \text{triangle law} \} \\
& h \cdot id_{list\,X} \\
= & \\
& h.
\end{aligned}$$

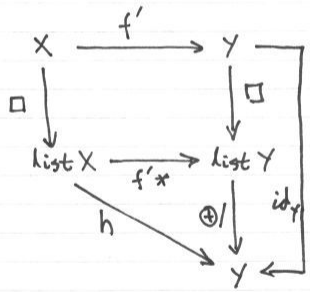


Although this argument is expressed in the notation of the list data type, it /pto

relies only on properties that hold in any adjunction, so there is a corresponding homomorphism theorem for every data type which is an adjunction.

To see that f is unique, suppose $h = \oplus/ \cdot f'*$. Then

$$
\begin{aligned}
& f \\
= & \{ \text{def } f \} \\
& h \cdot \Box \\
= & \{ \text{hyp} \} \\
& \oplus/ \cdot f'* \cdot \Box \\
= & \{ \Box \text{ natural} \} \\
& \oplus/ \cdot \Box \cdot f' \\
= & \{ \text{triangle law} \} \\
& id_Y \cdot f' \\
= & \\
& f'
\end{aligned}
$$



so $f = f'$.

Again this argument works for any adjunction

5.9

Example

Consider the type btree X defined by

```
btree α ::= Tip α
          | Fork (pair (btree α))
```

Following the type of Fork, we define a groupoid to be a pair $\langle X, f \rangle$ where X is a type and $f : \text{pair } X \longrightarrow X$. (Observe that $\text{Fork}_X : \text{pair}(\text{btree } X) \longrightarrow \text{btree } X$). A homomorphism of groupoids $h : \langle X, f \rangle \longrightarrow \langle Y, g \rangle$ is a function $h : X \longrightarrow Y$ s.t.

$$h \cdot f = g \cdot h^{(2)}$$

i.e. such that $h(f(x, y)) = g(h(x), h(y))$. The composition of two homomorphisms is again a homomorphism, and the identity function on X is a homomorphism from $\langle X, f \rangle$ to itself, so groupoids and homomorphisms form a category Groupoid.

There is a functor $U : \text{Groupoid} \longrightarrow \text{Type}$ defined by $U\langle X, f \rangle = X$ and

if $h : \langle X, f\rangle \longrightarrow \langle Y, g\rangle$ then $\mathcal{U}h = h$.

The type btree X provides a functor $F : \underline{\text{Type}} \longrightarrow \underline{\text{Groupoid}}$: we define

$$FX = \langle \text{btree } X,\ \text{Fork}_X \rangle$$

and if $f : X \longrightarrow Y$ then

$$Ff = f\circledast : FX \longrightarrow FY.$$

The function $f\circledast$ is a groupoid homomorphism because Fork is natural:

$$f\circledast \cdot \text{Fork} = \text{Fork} \cdot (f\circledast)^{(2)}$$

To make $\mathcal{U}$ and F into an adjunction we need to find natural transformations $\eta : \text{Id}_{\underline{\text{Type}}} \longrightarrow \mathcal{U}F$ and $\varepsilon : F\mathcal{U} \longrightarrow \text{Id}_{\underline{\text{Groupoid}}}$. For η we choose

$$\eta_X = \text{Tip}_X : X \longrightarrow \text{btree } X,$$

which we know to be natural. For ε we choose

$$\varepsilon_{\langle X,f\rangle} = f\curlywedge : \langle \text{btree } X, \text{Fork}_X\rangle \longrightarrow \langle X, f\rangle.$$

We need to check

* that $\varepsilon_{\langle X,f\rangle}$ is a homomorphism:

$$f\curlywedge(\text{Fork}(l,r)) = f(f\curlywedge l,\ f\curlywedge r)$$

* that $\varepsilon : F\mathcal{U} \longrightarrow \text{Id}_{\text{groupoid}}$ is natural: if $h : \langle X, f\rangle \longrightarrow \langle Y, g\rangle$ then

$$g\curlywedge \cdot f\circledast = h \cdot f\curlywedge$$

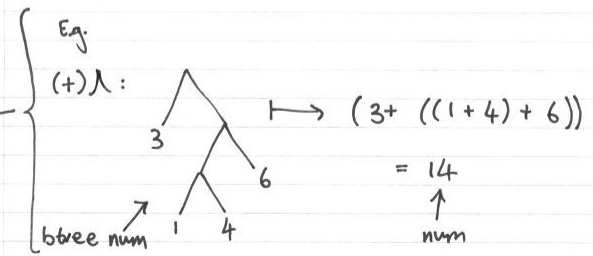
examples: $\text{Fork}\curlywedge \cdot f\circledast\circledast = f\circledast \cdot \text{Fork}\curlywedge$

$f\curlywedge \cdot \text{Fork}\curlywedge = f\curlywedge \cdot (f\curlywedge)\circledast$

Lastly, the two triangle laws must hold:

$$\mathcal{U}\varepsilon \cdot \eta\mathcal{U} = \text{Id}_{\mathcal{U}}$$

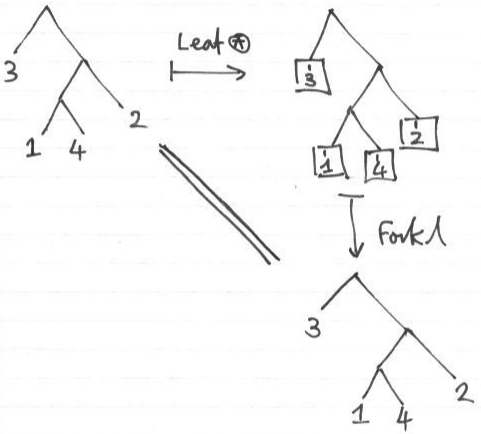
$$f\curlywedge \cdot \text{Leaf}_X = \text{id}_X$$



and

$$\varepsilon F \cdot F\eta = Id_F$$

$$Fork\wedge \cdot Leaf\circledast = id_{btree\,X}$$



So this is another adjunction, and there is a homomorphism theorem: every homomorphism $h : \langle btree\,X, Fork\rangle \longrightarrow \langle Y, f\rangle$ can be written uniquely as

$$h = f\wedge \cdot g\circledast$$

where $g = h \cdot Tip$.

As an application, consider the function

$$subst : (X \longrightarrow btree\,Y) \longrightarrow (btree\,X \longrightarrow btree\,Y)$$

such that subst σ applies the substitution σ to an "expression" represented by a btree. A recursive definition is

$$subst\ \sigma\ (Fork\ (l, r)) = Fork\ (subst\ \sigma\ l,\ subst\ \sigma\ r)$$

(i.e. subst σ is a homomorphism)

$$subst\ \sigma\ (Tip\ x) = \sigma\ x.$$

(i.e. subst $\sigma \cdot Tip = \sigma$)

The homomorphism theorem lets us write this concisely as

$$subst\ \sigma = Fork\wedge \cdot \sigma\circledast$$

5.10

Path trees

Consider the function pathtree of exercise 1.3:

$$pathtree : btree\ X \longrightarrow btree\ (list\ bool)$$

This is not a homomorphism from $\langle btree\,X, Fork\rangle$ to $\langle btree\,(list\ bool), Fork\rangle$, but we can make it a homomorphism by choosing a different operation on the right.

Observe first that pathtree is idempotent:
$\text{pathtree}_{\text{list bool}} \cdot \text{pathtree}_X = \text{pathtree}_X$
– and second that the pathtree of a tree depends only on those of its branches:

$$\text{pathtree}\ l = \text{pathtree}\ l' \wedge \text{pathtree}\ r = \text{pathtree}\ r' \Rightarrow \text{pathtree}(\text{Fork}(l,r)) = \text{pathtree}(\text{Fork}(l',r'))$$

Define $g(u,v) = \text{pathtree}(\text{Fork}(u,v))$
for u, v : btree (list bool)

Then

$$\begin{aligned} & \text{pathtree}(\text{Fork}(l,r)) \\ = & \quad \{\text{pathtree}\ l = \text{pathtree}(\text{pathtree}\ l)\ \text{etc.}\} \\ & \text{pathtree}(\text{Fork}(\text{pathtree}\ l, \text{pathtree}\ r)) \\ = & \quad \{\text{def } g\} \\ & g(\text{pathtree}\ u, \text{pathtree}\ v). \end{aligned}$$

So pathtree is a homomorphism

$$\langle \text{btree}\ X, \text{Fork} \rangle \longrightarrow \langle \text{btree (list bool)}, g \rangle,$$

and we can write

$$\text{pathtree} = g\Lambda \cdot f\circledast$$

where $f\,x = \text{pathtree}(\text{Tip}\ x) = \text{Tip}\ [\,]$
i.e. $f = \text{const}(\text{Tip}\ [\,])$.

Now we have to program g!

6/ Universals and uniqueness

The plan in this lecture is to show that an adjunction

$$\langle F, U, \eta, \varepsilon \rangle : \mathcal{X} \longrightarrow \mathcal{A}$$

is determined "up to isomorphism" by the functor U. This means that once we have fixed on a category $\mathcal{A}$ and related it to $\mathcal{X} = \text{Type}$ by a functor U, we have given a complete specification of a data type.

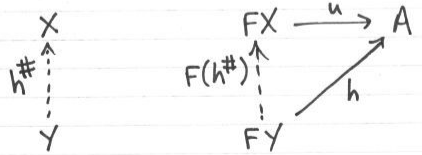
The program for carrying out this plan is slightly circuitous – we first introduce the notion of a universal arrow, examples of which are given by our "homomorphism theorems", and show that they are unique up to isomorphism. Next we show that any system of universal

arrows is generated by the homomorphism theorem for an adjunction unique up to isomorphism. It follows that the adjunction is uniquely determined by the data that determine the system of universal arrows, and that is just the functor U.

6.1 **Definition** If $\mathcal{X}$ and $\mathcal{A}$ are categories and $F: \mathcal{X} \longrightarrow \mathcal{A}$ a functor, then a universal arrow from F to $A \in \mathcal{A}$ is a pair $\langle X, u \rangle$ where $x \in X$ and $u: FX \longrightarrow A$:

$$X \qquad\qquad FX \xrightarrow{u} A$$

such that if $Y \in \mathcal{X}$ and $h: FY \longrightarrow A$ then there is a unique arrow $h^{\#}: Y \longrightarrow X$ in $\mathcal{X}$ s.t. $h = u \cdot F(h^{\#})$



□

The dotted arrows mean "this exists uniquely".

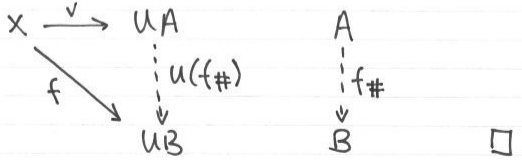
The homomorphism theorem tells us that $\langle UA, \varepsilon_A \rangle$ is a universal arrow from F to $A \in \mathcal{A}$ whenever $\mathcal{X}$ and $\mathcal{A}$ are related by an adjunction $\langle F, U, \eta, \varepsilon \rangle : \mathcal{X} \longrightarrow \mathcal{A}$

Dual to the notion of a universal arrow from $F: \mathcal{X} \longrightarrow \mathcal{A}$ to $A \in \mathcal{A}$ is the notion of a universal arrow from $X \in \mathcal{X}$ to $U: \mathcal{A} \longrightarrow \mathcal{X}$:

Definition A universal arrow from X to U is a pair $\langle A, v \rangle$ where $A \in \mathcal{A}$ and $v: X \longrightarrow UA$:

$$X \xrightarrow{v} UA \qquad\qquad A$$

such that if $B \in \mathcal{A}$ and $f: X \longrightarrow UB$ then there is a unique arrow $f^{\#}: A \longrightarrow B$ in $\mathcal{A}$ s.t. $f = U(f^{\#}) \cdot v$



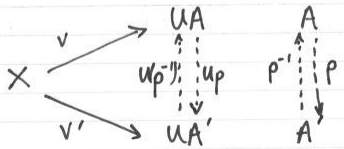
□

Reading the proof of the homomorphism theorem backwards gives a proof that $\langle FX, \eta_X \rangle$ is a universal arrow from X to U for every X.

It is this second notion of universal arrow that we shall use in this lecture, but everything we shall prove also holds by duality for our original version — for example, an adjunction
$\langle F, U, \eta, \varepsilon \rangle : \mathcal{X} \longrightarrow \mathcal{A}$
is uniquely determined up to isomorphism by the functor F.

6.2 Uniqueness of universals

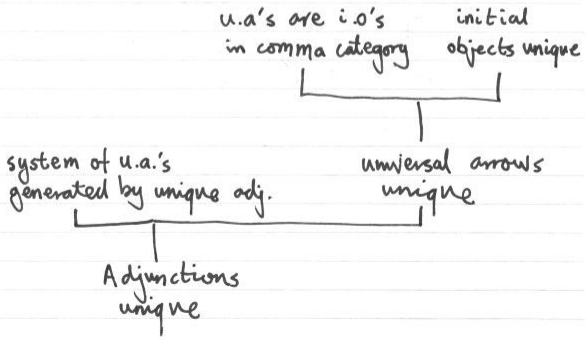
If $\langle A, v \rangle$ and $\langle A', v' \rangle$ are two universal arrows from X to U, then there is a unique isomorphism $p : A \longrightarrow A'$ s.t. $v' = Up \cdot v$ and $v = Up^{-1} \cdot v'$



Proof Define $p : A \longrightarrow A'$ by $p = v'^{\#}$: then $v' = Up \cdot v$ and p is unique with this property. Similarly define $p' : A' \longrightarrow A$ by $p' = v^{\#}$, so that $v = Up' \cdot v'$.

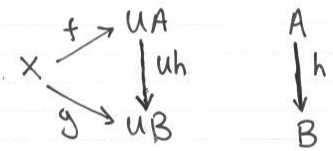
If we can show $p' \cdot p = id_A$ and $p \cdot p' = id_{A'}$, then p and $p' = p^{-1}$ will be the isomorphism we want. This is not too difficult to do directly, but instead we make a detour again, and use the "secret weapon" of category theory to transform the problem into the "prototypical

---

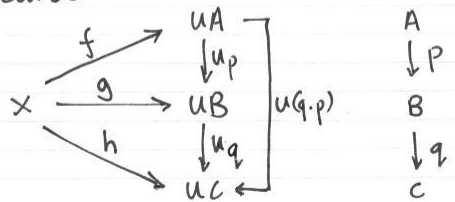


example" of uniqueness.

The secret weapon is the construction of a comma category $(X \downarrow U)$. The objects in this category are pairs $\langle A, f \rangle$ where $A \in \mathcal{A}$ and $f: X \to UA$ in $\mathcal{X}$. An arrow $h: \langle A, f \rangle \to \langle B, g \rangle$ in $(X \downarrow U)$ is an arrow $h: A \to B$ in $\mathcal{A}$ s.t. $g = Uh \cdot f$



Composition in $(X \downarrow U)$ is composition in $\mathcal{X}$ - the composition of two arrows of $(X \downarrow U)$ is another one because



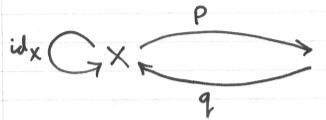
$$U(q \cdot p) \cdot f = Uq \cdot Up \cdot f = Uq \cdot g = h.$$

The first thing to notice is that a universal arrow $\langle A, v \rangle$ from X to U is an object in the category $(X \downarrow U)$. The second thing to notice in the definition of universal arrow is that if $\langle B, f \rangle$ is any other object in $(X \downarrow U)$ then there is a unique arrow $f^{\#}: \langle A, v \rangle \to \langle B, f \rangle$ in $(X \downarrow U)$. In other words, a universal arrow $\langle A, v \rangle$ is an initial object in $(X \downarrow U)$.

Definition An initial object X in a category $\mathcal{C}$ is an object $X \in \mathcal{C}$ s.t. if $Y \in \mathcal{C}$ is any (other) object, then there is a unique arrow $!_Y : X \to Y$ in $\mathcal{C}$. □

Proposition Any two initial objects X and X' in $\mathcal{C}$ are isomorphic

Proof Let $p = !_{X'} : X \to X'$ be the unique arrow from X to X'. Similarly, let $q = !'_X : X' \to X$ be the unique arrow from X' to X.



Now $q \cdot p$ is an arrow from X to X, so it must be equal to the arrow $id_X : X \to X$:

$$q \cdot p = id_X$$

Similarly, $p \cdot q = id_{X'}$, so p is an isomorphism with inverse $p^{-1} = q$. □

Corollary Universal arrows are unique up to an isomorphism in $(X \downarrow U)$.

6.3 Systems of universal arrows

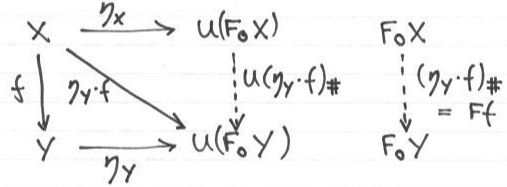
We've seen that every adjunction gives us a system of universal arrows $\langle FX, \eta_X \rangle$, one for each $X \in \mathcal{X}$. We now prove the converse - that each such system of universal arrows is given by a unique adjunction:

Suppose that for each $X \in \mathcal{X}$, $\langle F_0 X, \eta_X \rangle$ is a universal arrow from X to U. Then there is a unique adjunction

$$\langle F, U, \eta, \varepsilon \rangle : \mathcal{X} \longrightarrow \mathcal{A}$$

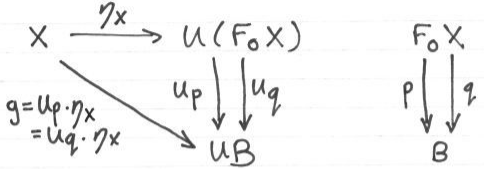
such that $FX = F_0 X$ for each $X \in \mathcal{X}$.

Proof We define F on arrows as follows: if $f : X \to Y$ in $\mathcal{A}$



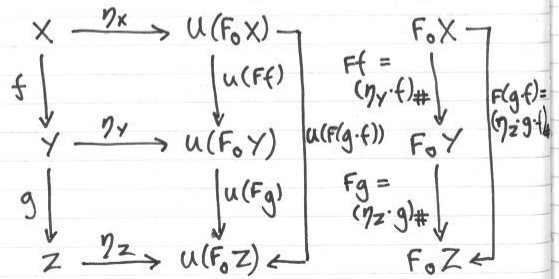
we define $Ff = (\eta_Y \cdot f)_\# : F_0 X \to F_0 Y$ i.e. the unique arrow $h : F_0 X \to F_0 Y$ s.t. $\eta_Y \cdot f = Uh \cdot \eta_X$. This does define a functor: for instance $F(g \cdot f) = Fg \cdot Ff$. We prove this as follows: both $F(g \cdot f)$ and $Fg \cdot Ff$ are arrows $F_0 X \to B$ in $\mathcal{A}$ (where $B = F_0 Z$ in this case).

A general strategy for proving the equality of two such functions $p, q : F_0 X \longrightarrow B$ in $\mathcal{A}$ is this:



we show that $U_p \cdot \eta_X = U_q \cdot \eta_X$
It follows that $p = q_\# = q$, where $g : X \longrightarrow UB$ is the arrow defined by $g = U_p \cdot \eta_X = U_q \cdot \eta_X$, because there is a unique function $g_\#$ with the property $g = U(g_\#) \cdot \eta_X$.

In the case of $F(g \cdot f)$ and $Fg \cdot Ff$,



we argue as follows:

$$
\begin{aligned}
& U(F(g \cdot f)) \cdot \eta_X \\
= & \quad \{ \text{def of } F \} \\
& U(\eta_Z \cdot g \cdot f)_\# \cdot \eta_X \\
= & \quad \{ \text{def of } \# \} \\
& \eta_Z \cdot g \cdot f \\
= & \quad \{ \text{def of } \# \} \\
& U(\eta_Z \cdot g)_\# \cdot \eta_Y \cdot f \\
= & \quad \{ \text{def of } \# \} \\
& U(\eta_Z \cdot g)_\# \cdot U(\eta_Y \cdot f)_\# \cdot \eta_X \\
= & \quad \{ \text{def of } F \} \\
& U(Fg) \cdot U(Ff) \cdot \eta_X
\end{aligned}
$$

Therefore $F(g \cdot f) = Fg \cdot Ff$.

The definition of η we have already; its naturality comes from the definition of F:

$$
\begin{aligned}
U(Ff) \cdot \eta_X &= U(\eta_Y \cdot f)_\# \cdot \eta_X \\
&= \eta_Y \cdot f .
\end{aligned}
$$